Object-Oriented Programming

Task 1: Class and Object Creation

In [9]:
```python
# Defining the Vehicle class
class Vehicle:
    # Initializing with attributes
    def __init__(self, vehicle_name, vehicle_type, max_speed):
        self.vehicle_name = vehicle_name
        self.vehicle_type = vehicle_type
        self.max_speed = max_speed

    # Method to display information about the vehicle
    def display_info(self):
        return f"Vehicle Name: {self.vehicle_name}, Type: {self.vehicle_type}, Max

    # Method to simulate the vehicle honking
    def honk(self):
        return f"{self.vehicle_name} goes 'Beep Beep!'"

# Creating a car
car = Vehicle("Lamborghini", "Car", 350)
print(car.display_info())
print(car.honk())

# Creating a motorcycle
motorcycle = Vehicle("Harley Davidson", "Motorcycle", 150)
print(motorcycle.display_info())
print(motorcycle.honk())
```

```
Vehicle Name: Lamborghini, Type: Car, Max Speed: 350 km/h
Lamborghini goes 'Beep Beep!'
Vehicle Name: Harley Davidson, Type: Motorcycle, Max Speed: 150 km/h
Harley Davidson goes 'Beep Beep!'
```

Task 2: Inheritance

In [12]:
```python
# Defining the main class named "Vehicle"
class Vehicle:
    # Initializing with attributes
    def __init__(self, vehicle_name, vehicle_type, max_speed):
        self.vehicle_name = vehicle_name
        self.vehicle_type = vehicle_type
        self.max_speed = max_speed

    # Method to display information about the vehicle
    def display_info(self):
        return f"Vehicle Name: {self.vehicle_name}, Type: {self.vehicle_type}, Max

# Defining the sub-class named "Car"
class Car(Vehicle):

    # Method to simulate the vehicle honking
```

```
    def honk(self):
        return f"{self.vehicle_name} goes 'Beep Beep!'"

#Displaying all the information from both classes
car = Car("G-Wagon", "Car", 220)
print(car.display_info())
print(car.honk())
```

```
Vehicle Name: G-Wagon, Type: Car, Max Speed: 220 km/h
G-Wagon goes 'Beep Beep!'
```

Task 3: Polymorphism

In [17]:

```
# Defining the main class named "Vehicle"
class Vehicle:
    # Initializing with attributes
    def __init__(self, vehicle_name, vehicle_type, max_speed):
        self.vehicle_name = vehicle_name
        self.vehicle_type = vehicle_type
        self.max_speed = max_speed

    # Method to display information about the vehicle
    def display_info(self):
        return f"Vehicle Name: {self.vehicle_name}, Type: {self.vehicle_type}, Max

# Defining the subclass "Car" that inherits from "Vehicle"
class Car(Vehicle):
    # Initializing with attributes, calling the parent class initializer
    def __init__(self, vehicle_name, max_speed):
        super().__init__(vehicle_name, "Car", max_speed)

    # Overriding the display_info method
    def display_info(self):
        return f"{self.vehicle_name} (Car) goes 'Beep Beep!'"

# Function to demonstrate polymorphism
def vehicle_info(vehicle):
    print(vehicle.display_info())

# Creating instances of Vehicle and Car
my_vehicle = Vehicle("Generic Vehicle", "Truck", 120)
my_car = Car("Toyota Corolla", 180)

# Calling vehicle_info with different types
vehicle_info(my_vehicle)
vehicle_info(my_car)
```

```
Vehicle Name: Generic Vehicle, Type: Truck, Max Speed: 120 km/h
Toyota Corolla (Car) goes 'Beep Beep!'
```

Task 4: Encapsulation

In [10]:

```
# Defining the main class named "Car"
class Car:
    # Initializing with attributes
    def __init__(self, vehicle_name, vehicle_type, max_speed):
```

```
        self.vehicle_name = vehicle_name
        self.vehicle_type = vehicle_type
        self.max_speed = max_speed
        self.__private_var = "I am private"

    # Private method to display information about the vehicle
    def __display_info(self):
        return f"Vehicle Name: {self.vehicle_name}, Type: {self.vehicle_type}, Max

    # Public method to access the private method
    def get_display_info(self):
        return self.__display_info()

# Creating an instance of Car
car = Car("Generic Vehicle", "Truck", 120)
print(car.get_display_info())
```

```
Vehicle Name: Generic Vehicle, Type: Truck, Max Speed: 120 km/h
```

Task 5: Abstraction

In [1]:
```
# Importing important libraries for abstraction
from abc import ABC,abstractmethod
# Importing math library for mathematical calculations
import math

# Defining the main class "Shape"
class Shape(ABC):
    @abstractmethod
    def area(self):
        #pass function is used to pass the charactersitics from parent class to chi
        pass

# Defining the sub class "Circle"
class Circle(Shape):
    def __init__(self,radius):
        self.radius = radius
    def area(self):
        return 3.14*(self.radius**2)

# Defining the sub class "Rectangle"
class Rectangle(Shape):
    def __init__(self,width,height):
        self.width=width
        self.height = height
    def area(self):
        return self.width * self.height

circle = Circle(5)
rectangle = Rectangle(4,5)
print("Area of circle",circle.area())
print("Area of rectangle",rectangle.area())
```

```
Area of circle 78.5
Area of rectangle 20
```

Reflection on the Role of OOP Principles in Code Organization and Structure

Object-Oriented Programming (OOP) principles played a crucial role in organizing and structuring the code, making it easier to manage, understand, and extend. Here's a brief reflection on how each OOP principle contributed:

1. Encapsulation: By grouping related data and methods within classes, encapsulation helped create self-contained units, like `Vehicle`, `Car`, and `Dog`. Each class manages its own data and behavior, making the code modular and reducing dependencies. This encapsulation ensures that details of each class's implementation remain hidden, exposing only what is necessary through public methods. As a result, it was easy to make changes within a class without affecting the rest of the program.

2. Abstraction: OOP allowed me to define classes that represent real-world concepts without needing to delve into the details each time I used them. For example, in a `Vehicle` class, I could define high-level methods like `start_engine()` or `display_info()` without worrying about the specific internal mechanics. This abstraction made it easier to work with complex objects, as only the relevant attributes and methods needed to be accessed, keeping the code clear and focused on higher-level functionality.

3. Inheritance: Inheritance was invaluable for creating subclasses that shared common behavior but also had their own specific attributes or methods. For instance, a `Car` class could inherit from `Vehicle`, meaning I didn't need to duplicate properties like `vehicle_type` and `max_speed` but could still add unique behavior specific to cars. Inheritance enabled code reusability and reduced redundancy, making it simpler to introduce new types of vehicles without rewriting shared functionality.

4. Polymorphism: Through polymorphism, I could define a common interface for different classes, allowing them to be used interchangeably. For instance, both `Vehicle` and `Car` could have a `display_info()` method, but each could provide a distinct implementation. This flexibility meant I could interact with different objects through a shared interface, making the code more flexible and easier to extend.

Conclusion

Using OOP principles, the code became organized, modular, and easier to understand. Each class has a clear purpose and scope, with the ability to interact smoothly with other parts of the program. Moreover, the code is well-structured for potential future expansion. OOP not only improved code readability and reusability but also enhanced maintainability, making it easier to identify and fix issues or add new features with minimal impact on the existing code.